**Pune Institute of Computer Technology Dhankawadi, Pune**

# Mini-Project

String Matching Algorithms

**SUBMITTED BY**

**Devansh Mundada (41253)**

**Karishma Nair (41254)**

**Ved Patwardhan (41260)**

**Yash Pawar (41261)**

**Under the guidance of**

Prof. Yogesh Handge



DEPARTMENT OF COMPUTER ENGINEERING

**Academic Year 2022-23**

**PUNE INSTITUTE OF COMPUTER TECHNOLOGY,DHANKAWADI PUNE- 43.**

**PUNE INSTITUTE OF COMPUTER TECHNOLOGY, DHANKAWADI PUNE-43.**

**CERTIFICATE**



This is to certify that **Devansh Mundada (41253), Karishma (41254), Ved Patwardhan (41260), Yash Pawar (41261)** student of B.E. (Computer Engineering Department) Batch 2022-2023, have satisfactorily completed a report on “**String Matching Algorithms**” towards the partial fulfillment of the fourth year Computer Engineering Semester VII of SPPU.

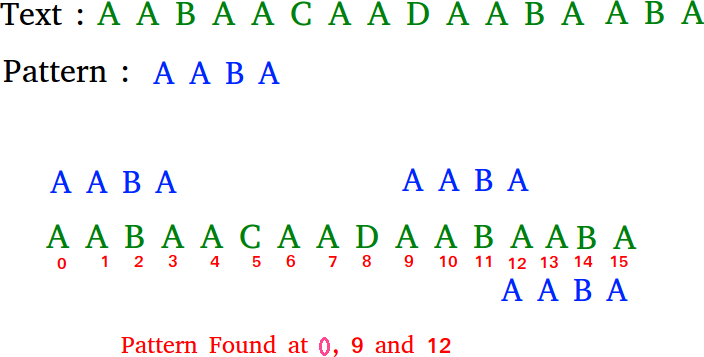
**Date: Place:**

Dr. G. V. Kale **Head of Department*,* Computer Engineering**

**String Matching Algorithm**

# Introduction

In computer science, string-searching algorithms, sometimes called string- matching algorithms, are an important class of string algorithms that try to find a place where one or several strings (also called patterns) are found within a larger string or text.



Pattern searching is an important problem in computer science. When we do search for a string in a notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

# Problem Statement

Implement the Naive string-matching algorithm and Rabin-Karp algorithm for string matching. Observe difference in working of both the algorithms for the same input.

# Objective

To understand the difference between the working of Naive string-matching algorithm and Rabin-Karp algorithm. Also compare the time complexity of the algorithms

# Theory

**ľhe “Naive” Method**

Its idea is straightforward — for every position in the text, consider it a starting position of the pattern and see if you get a match. The “naive” approach is easy to understand and implement but it can be too slow in some cases. If the length of the text is n and the length of the pattern m, in the worst case it may take as much as (n \* m) iterations to complete the task.

Algorithm:

function brute\_force(text[], pattern[]) {

// let n be the size of the text and m the size of the

// pattern

for (i = 0; i < n; i++) {

for (j = 0; j < m && i + j < n; j++)

if (text[i + j] != pattern[j]) break;

// mismatch found, break the inner loop if (j == m) // match found

}

}

**Rabin Kaíp Algoíithm**

This is the “naive” approach augmented with a powerful programming technique – the hash function. Every string s[] of length m can be seen as a number H written in a positional numeral system in base B (B >= size of the alphabet used in the string):

H = s[0] \* B(m – 1) + s[1] \* B(m – 2) + … + s[m - 2] \* B1 + s[m - 1] \* B0

If we calculate the number H (the hash value) for the pattern and the same number for every substring of length m of the text than the inner loop of the “naive” method will disappear – instead of comparing two strings character by character we will have just to compare two integers.

Algorithm

// correctly calculates a mod b even if a < 0 function int\_mod(int a, int b) {

return (a % b + b) % b;

}

function Rabin\_Karp(text[], pattern[]) {

// let n be the size of the text, m the size of the

// pattern, B - the base of the numeral system,

// and M - a big enough prime number

if (n < m) return; // no match is possible

// calculate the hash value of the pattern hp = 0;

for (i = 0; i < m; i++)

hp = int\_mod(hp \* B + pattern[i], M);

// calculate the hash value of the first segment

// of the text of length m ht = 0;

for (i = 0; i < m; i++)

ht = int\_mod(ht \* B + text[i], M);

if (ht == hp) check character by character if the first

segment of the text matches the pattern;

// start the "rolling hash" - for every next character in

// the text calculate the hash value of the new segment

// of length m; E = (Bm-1) modulo M for (i = m; i < n; i++) {

ht = int\_mod(ht - int\_mod(text[i - m] \* E, M), M); ht = int\_mod(ht \* B, M);

ht = int\_mod(ht + text[i], M);

if (ht == hp) check character by character if the

current segment of the text matches the pattern;

}

}

**CODE :**

**ľhe “Naive” Method**

#include <bits/stdc++.h>

**using namespace** std;

**void** search(**char**\* pat, **char**\* txt)

{

**int** M = **strlen**(pat); **int** N = **strlen**(txt);

/\* A loop to slide pat[] one by one \*/

**for** (**int** i = 0; i <= N - M; i++) {

**int** j;

/\* For current index i, check for pattern match \*/

**for** (j = 0; j < M; j++)

**if** (txt[i + j] != pat[j])

**break**;

**if** (j

== M) // if pat[0...M-1] = txt[i, i+1, ...i+M-1] cout << "Pattern found at index " << i << endl;

}

}

// Driver's Code

**int** main()

{

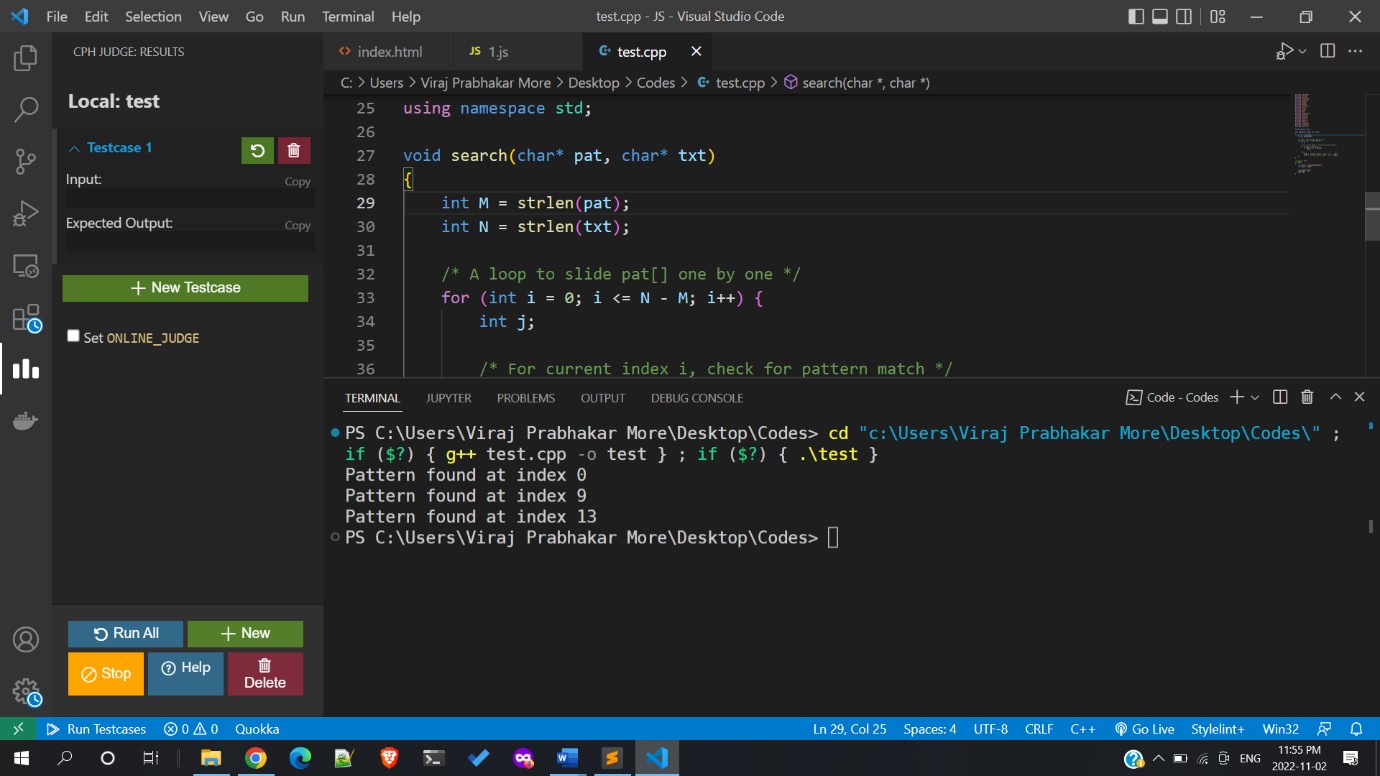
**char** txt[] = "AABAACAADAABAAABAA";

**char** pat[] = "AABA";

// Function call search(pat, txt); **return** 0;

}

**OUTPUT**



Time Complexity: O(N2) Auxiliary Space: O(1)

**Rabin Kaíp**

#include <bits/stdc++.h> **using namespace** std; #define d 256

**void** search(**char** pat[], **char** txt[], **int** q)

{

**int** M = **strlen**(pat); **int** N = **strlen**(txt); **int** i, j;

**int** p = 0; // hash value for pattern

**int** t = 0; // hash value for txt

**int** h = 1;

**for** (i = 0; i < M - 1; i++) h = (h \* d) % q;

**for** (i = 0; i < M; i++) {

p = (d \* p + pat[i]) % q; t = (d \* t + txt[i]) % q;

}

**for** (i = 0; i <= N - M; i++) {

**if** (p == t) {

**for** (j = 0; j < M; j++) {

**if** (txt[i + j] != pat[j]) {

**break**;

}

}

**if** (j == M)

cout << "Pattern found at index " << i

<< endl;

}

**if** (i < N - M) {

t = (d \* (t - txt[i] \* h) + txt[i + M]) % q;

**if** (t < 0)

t = (t + q);

}

}

}

**int** main()

{

**char** txt[] = "FUNNY AND FUNNY";

**char** pat[] = "FUN";

**int** q = INT\_MAX; search(pat, txt, q);

**return** 0;

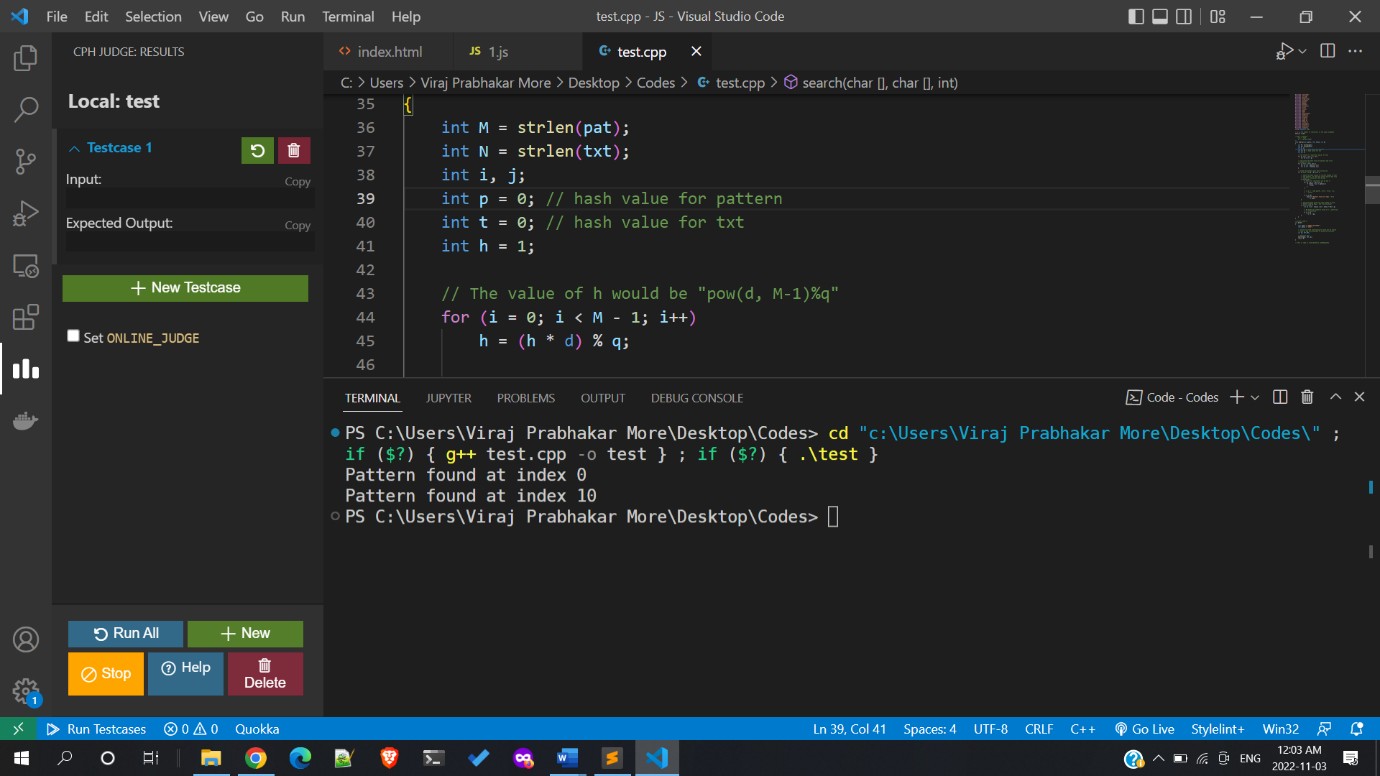
}

Time Complexity:

The average and best-case running time of the Rabin-Karp algorithm is O(n+m), but its worst-case time is O(nm).

Auxiliary Space: O(1)

**OUľPUľ:**



# Conclusion

Thus, we have implemented and compared time complexity and analysed performance of Naive string-matching algorithm and Rabin-Karp algorithm.

1. TopCoder

# References

https://[www.topcoder.com/thrive/articles/Introduction%20to%20String%](http://www.topcoder.com/thrive/articles/Introduction%20to%20String%25) 20Searching%20Algorithms

1. Geeks For Geeks:

https://[www.geeksforgeeks.org/algorithms-gq/pattern-searching/?ref=lbp](http://www.geeksforgeeks.org/algorithms-gq/pattern-searching/?ref=lbp)

1. University of Auckland https://[www.cs.auckland.ac.nz/courses/compsci369s1c/lectures/GG-](http://www.cs.auckland.ac.nz/courses/compsci369s1c/lectures/GG-) notes/CS369-StringAlgs.pdf